



Smooth Heaps and a Dual View of Self-Adjusting Data Structures

László Kozma

Eindhoven University of Technology
Netherlands

lkozma@gmail.com

Thatchaphol Saranurak

KTH Royal Institute of Technology, Stockholm
Sweden

thasar@kth.se

ABSTRACT

We present a new connection between self-adjusting binary search trees (BSTs) and heaps, two fundamental, extensively studied, and practically relevant families of data structures (Allen, Munro, 1978; Sleator, Tarjan, 1983; Fredman, Sedgwick, Sleator, Tarjan, 1986; Wilber, 1989; Fredman, 1999; Iacono, Özkan, 2014). Roughly speaking, we map an *arbitrary* heap algorithm within a broad and natural model, to a corresponding BST algorithm with the same cost on a *dual* sequence of operations (i.e. the same sequence with the roles of *time* and *key-space* switched). This is the first general transformation between the two families of data structures.

There is a rich theory of dynamic optimality for BSTs (i.e. the theory of competitiveness between BST algorithms). The lack of an analogous theory for heaps has been noted in the literature (e.g. Pettie; 2005, 2008). Through our connection, we transfer all instance-specific lower bounds known for BSTs to a general model of heaps, initiating a theory of *dynamic optimality for heaps*.

On the algorithmic side, we obtain a new, simple and efficient heap algorithm, which we call the *smooth heap*. We show the smooth heap to be the heap-counterpart of Greedy, the BST algorithm with the strongest proven and conjectured properties from the literature, widely believed to be instance-optimal (Lucas, 1988; Munro, 2000; Demaine et al., 2009). Assuming the optimality of Greedy, the smooth heap is also optimal within our model of heap algorithms.

Intriguingly, the smooth heap, although derived from a non-practical BST algorithm, is simple and easy to implement (e.g. it stores no auxiliary data besides the keys and tree pointers). It can be seen as a variation on the popular *pairing heap* data structure, extending it with a “power-of-two-choices” type of heuristic. For the smooth heap we obtain instance-specific upper bounds, with applications in adaptive sorting, and we see it as a promising candidate for the long-standing question of a simpler alternative to Fibonacci heaps.

The paper is dedicated to Raimund Seidel on occasion of his sixtieth birthday.

CCS CONCEPTS

• Theory of computation → Data structures design and analysis; Sorting and searching;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

STOC’18, June 25–29, 2018, Los Angeles, CA, USA

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

ACM ISBN 978-1-4503-5559-9/18/06...\$15.00

<https://doi.org/10.1145/3188745.3188864>

KEYWORDS

heaps, binary search trees, self-adjusting data structures, sorting

ACM Reference Format:

László Kozma and Thatchaphol Saranurak. 2018. Smooth Heaps and a Dual View of Self-Adjusting Data Structures. In *Proceedings of 50th Annual ACM SIGACT Symposium on the Theory of Computing (STOC’18)*. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3188745.3188864>

1 INTRODUCTION

We revisit¹ two popular families of comparison-based data structures that implement the fundamental *dictionary* and *priority queue* abstract data types. A **dictionary** stores a set of keys from an ordered universe, supporting the operations *search*, *insert*, and *delete*. A **priority queue** stores a set of keys from an ordered universe (often referred to as priorities), with typical operations such as finding and deleting the minimum key (“extract-min”), inserting a new key, decreasing a given key, and merging two priority queues. The fact that efficient search is *not* required allows priority queues to perform some operations faster than dictionaries.

Perhaps the best known and most natural implementations of dictionaries and priority queues are binary search trees (BSTs), respectively, multiary heaps. In both cases, a key is associated to each node of the underlying tree structure, according to the standard in-order, respectively, (min-)heap order conditions.

Both structures need, in order to remain efficient over the long term, an occasional re-structuring. In BSTs, the costs of the main operations are proportional to the depths of the affected nodes, the goal is therefore to keep the tree reasonably balanced. For heaps, the desired structure is, in some sense, the opposite; the crucial extract-min operations are easiest to perform if the heap is fully sorted, i.e. if it is a path. The design and analysis of efficient algorithms for heap- and BST maintenance continues to be a central topic of algorithmic research, and a large number of ingenious techniques have been proposed in the literature.

Our contribution is a new connection between the two tasks, showing that (with some reasonable restrictions), an arbitrary algorithm for heap maintenance encodes an algorithm for BST maintenance on a related (dual) sequence of inputs. This is the first general transformation between the two families of data structures. The connection allows us to transfer results (both algorithms and lower bounds) between the two settings.

On the algorithmic side, we obtain a new, simple and efficient heap data structure, which we call the *smooth heap*. We show the smooth heap to be the heap-equivalent of Greedy, the BST algorithm with the strongest proven and conjectured properties from the literature. We thus obtain, for the smooth heap, instance-specific guarantees that go beyond the logarithmic amortized guarantees

¹A longer, preprint version of this paper is available at <http://arxiv.org/abs/1802.05471>.

known for existing heap implementations. Key to our result is a new, non-deterministic interpretation of Greedy [43, § 6], which may be of independent interest.

On the complexity side, we transfer all instance-specific lower bounds known for BSTs to a natural class of heap algorithms. Pettie observed [54, 55] that a theory of dynamic optimality for heaps, analogous to the rich set of results known for BSTs, is missing. We see our instance-specific lower and upper bounds as essential components of such a theory.

Self-adjusting BSTs. The study of dictionaries based on BSTs goes back to the beginnings of the field. Standard balanced trees such as AVL-trees [1], red-black trees [3, 28], or randomized treaps [57] keep the tree balanced by storing and enforcing explicit bookkeeping information. We refer to Knuth [42, § 6.2] for an extensive overview of classical methods.

By contrast, splay trees, invented by Sleator and Tarjan [58], do not store any auxiliary information (either in the nodes of the tree or globally) besides the keys of the dictionary and the pointers representing the tree. Instead, they employ an elegant local restructuring whenever a key is accessed. In short, splay trees bring the accessed key to the root via a sequence of *double-rotations*. Splay trees can be seen as a member of a more general family of *self-adjusting* data structures.

Splay trees support all main operations in (optimal) logarithmic amortized time. Furthermore, they adapt to various usage patterns, providing stronger than logarithmic guarantees for certain structured sequences of operations; we call such guarantees *instance-specific*. Several instance-specific upper bounds are known for splay trees (see e.g. [9, 10, 15, 55, 58, 60, 62]) and others have been conjectured. In fact, Sleator and Tarjan conjectured splay trees to be *instance-optimal*, i.e. to match, up to a constant factor, the optimum among all BST algorithms.² This is the famous *dynamic optimality conjecture*, still open. It is not known whether *any* BST algorithm has such a property, even with a priori knowledge of the entire sequence of operations.

Lucas [47] proposed, as a theoretical construction, a BST algorithm that takes into account *future* operations; in her algorithm, the search path is re-arranged such as to bring the soonest-to-be-accessed key to the root (and similarly in a recursive way in the left and right subtrees of the root). Following Demaine, Harmon, Iacono, Kane, and Pătraşcu [11], we refer to this algorithm as GreedyFuture. GreedyFuture is widely believed to be $O(1)$ -competitive [11, 47, 51], and is known to match essentially all instance-specific guarantees proven for splay trees, as well as some stronger guarantees not known to hold for splay trees [6, 35]. Unfortunately, $o(\log n)$ -competitiveness has not been shown for either Splay or GreedyFuture.

It was shown in [11] that GreedyFuture can be simulated by an *online* algorithm (which we refer to simply as Greedy), with only a constant factor increase in cost. In a different line of work, Demaine et al. described an online BST algorithm with a competitive ratio of $O(\log \log n)$, called Tango tree [12].

²The term “instance-optimal” is, in our context, equivalent with “ $O(1)$ -competitive”. An algorithm is c -competitive, if its running time is at most c times the running time of the optimal algorithm (for every given input).

Self-adjusting heaps. In 1984, Fredman and Tarjan introduced the Fibonacci heap [24], a data structure that achieves the theoretically optimal amortized bounds of $O(\log n)$ for extract-min, and $O(1)$ for all other heap operations. Fibonacci heaps are rather complicated to implement and simpler alternatives tend to outperform them in practice [44]. In the past three decades, a central goal of research in data structures has thus been to find a simpler heap implementation that would match the theoretical guarantees of Fibonacci heaps (see e.g. [4, 5, 8, 14, 17, 30, 31, 38, 59]).

Arguably, this goal has not been fully achieved. In particular, Fibonacci heaps, as well as most of their proposed alternatives store auxiliary bookkeeping information, or perform operations outside the comparison/pointer model, or maintain a pointer structure that is not “forest-like” (not following, as Iacono and Özkan [37] put it, “the spirit of what we usually think of as a heap”). The following general question is still open.

Question 1. *Is there, by analogy to search trees, a simple, self-adjusting heap data structure with optimal amortized cost, possibly achieving instance-optimality?*

The closest in simplicity and elegance to self-adjusting trees are *pairing heaps*, introduced by Fredman, Sedgewick, Sleator, and Tarjan [23]. Pairing heaps do not store auxiliary information besides the keys and the pointers of a single (multi-way) heap, are easy to implement and efficient in practice [44]. They implement all operations using the unit-cost *link* primitive (Figure 1). Pairing heaps perform extract-min, the only operation with a non-trivial implementation, in a *two-pass re-structuring* of the children of the root.

Fredman et al. showed that pairing heaps support all operations in logarithmic amortized time. They also conjectured that pairing heaps match the optimal bounds of Fibonacci heaps. This conjecture was disproved by Fredman [21], who showed that in certain sequences, *decrease-key* may cost $\Omega(\log \log n)$. A similar lower bound was later shown by Iacono and Özkan [36, 37].

The Fredman-, and Iacono-Özkan lower bounds hold for broad classes of “pairing-heap-like” data structures that include all natural variants of pairing heaps, as well as some other heap data structures from the literature. The heap models defined in these works are “in the spirit” of self-adjusting trees; to the extent that these models capture the idea of a “self-adjusting heap”, the answer to Question 1 has to be negative. We argue, however, that there exist natural self-adjusting heaps that fall outside these models, leaving Question 1 open.

For some concrete heaps, instance-specific *upper bounds* were studied by Iacono and Langerman [34], Elmasry [16], and Elmasry et al. [18]. On the other hand, a theory of instance-specific *lower bounds* (in a sense similar to BSTs) has not yet been proposed for heaps in *any heap model*.

Connecting BSTs and heaps. Self-adjusting BSTs and heaps serve different purposes, and the underlying trees they maintain have, in general, different characteristics. Yet, they are similar in their approach of performing local re-adjustments (using rotations, respectively links), seemingly ignoring global structure. The two families of data structures have been introduced and studied in the past decades by largely the same authors.

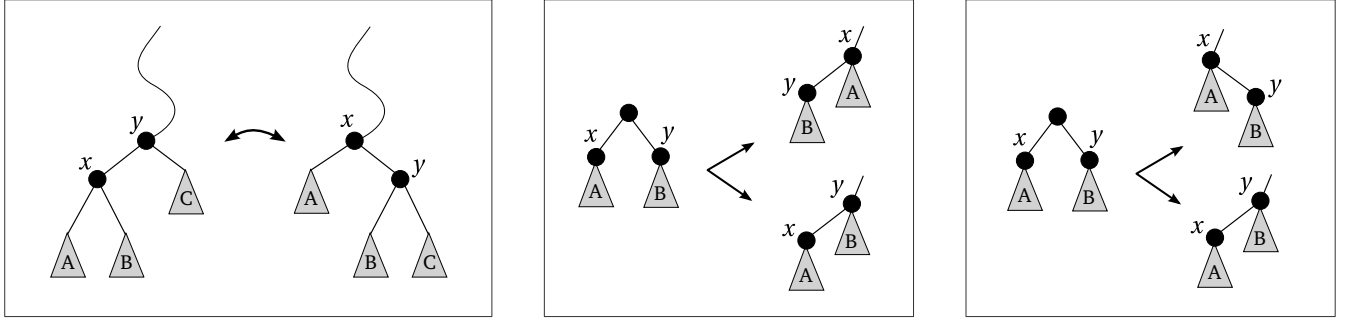


Figure 1: Elementary operations in BSTs and heaps. Letters x and y denote keys, A, B, C are arbitrary subtrees. (i) Rotation in a binary search tree. (ii) Classical (“unstable”) link between neighboring siblings in a heap. The larger of x and y becomes the leftmost child of the smaller. (iii) Stable link between neighboring siblings in a heap. The larger of x and y becomes the leftmost child of the smaller, if it was originally to the left; otherwise it becomes the rightmost child.

The following technical connection between splay trees and the standard variant of pairing heaps has been observed by Fredman et al. [23]. If we view a heap as a binary tree (interpreting “leftmost-child” and “right-sibling” pointers as “left-child” and “right-child”, see e.g. [41, § 2.3.2]), the re-structuring of pairing heaps after an extract-min operation resembles the re-adjustment of splay trees during an access. Using this observation, Fredman et al. adapt the splay tree potential function [58] to show that pairing heap operations take logarithmic time. Despite the productive use of this connection, it seems rather specific to splay trees and pairing heaps. Given the intriguing similarity between self-adjusting data structures, the following question suggests itself.

Question 2. *Is there a fundamental, general correspondence between self-adjusting BST and self-adjusting heap data structures?*

1.1 Our Results

We propose a general model of heaps, which we call the *stable heap model*.³ This model allows us to approach both **Question 1** and **Question 2** in surprising ways. The model aims to capture self-adjusting heap algorithms, and is somewhat similar to the “pure heap” model of Iacono and Özkan [36, 37].

The new element of our model is the *stable link* operation that replaces the *link* operation of existing heap models. Whereas the standard link operation makes the larger of the two linked items the leftmost child of the smaller, our stable link makes the larger item either the leftmost or the rightmost child of the smaller (see Figure 1). More precisely, if x and y are linked, where x is the left neighbor of y , then, a stable link operation makes x the *leftmost* child of y , or it makes y the *rightmost* child of x , respecting the (min-)heap order condition.

An intuitive a priori reason for stable links is that they “better preserve” the original left-to-right ordering of keys, which we expect to make algorithms that use stable links more amenable to

instance-specific analysis.⁴ A second justification is that the lower bounds known in existing heap models [21, 36, 37] crucially exploit that the links are not stable, raising the possibility that an algorithm that uses stable links may circumvent the existing lower bounds. The main motivation, however, is that with stable links, the heap model turns out to be deeply connected to the BST model.

General transformation. Our connection addresses **Question 2** as follows. We show that within the stable heap model, every algorithm $\mathcal{A}$ for heap re-structuring corresponds to some algorithm $\mathcal{B}$ for BST re-arrangement. This is the first general connection between models of heaps and BSTs. The costs of the two algorithms in *sorting-mode* may differ only by a constant factor, when executing “dual” sequences of operations. By *sorting-mode* execution we mean the following (see Figure 2 for illustration).

Sorting-mode for a heap: A sequence of n keys $(x_1, \dots, x_n)$ is inserted into an initially empty heap, followed by n extract-min operations. For simplicity, assume that $\{x_1, \dots, x_n\} = [n]$. The execution can be interpreted as sorting the permutation $(x_1, \dots, x_n)$ via *selection-sort*, using a particular heap-based method for selecting the minimum.

Sorting-mode for a BST: A sequence of n keys $(x_1, \dots, x_n)$ is inserted into an originally empty BST. Again, assume $\{x_1, \dots, x_n\} = [n]$. The execution can be interpreted as sorting $(x_1, \dots, x_n)$ via *insertion-sort*, using a particular BST-based method for insertion. (The sorted keys can be read out in an $O(n)$ -time traversal of the final tree.)

The connection between $\mathcal{A}$ and $\mathcal{B}$ is the following. The number of elementary operations performed by $\mathcal{A}$ when sorting $X = (x_1, \dots, x_n)$ equals (up to a constant factor) the number of elementary operations performed by $\mathcal{B}$ when sorting the *inverse permutation* X' , i.e. the permutation $X' = (y_1, \dots, y_n)$, where $y_i = j$ iff $x_j = i$. (See Figure 2 for an illustration.)

We say that the roles of *time* and *key-space* are switched between the two problems. The time of insertion, a.k.a. the *index* of a key in the heap input is mapped to the *rank* of a key in the BST input.

³By *model* we understand the description of the pointer-based structure and the set of primitive operations that may be used to implement the priority queue.

⁴The name is inspired by the idea of “stable sorting” (see e.g. [42, § 5]). Stable sorting preserves the original ordering of key-pairs that are *equal*. As seen later, stable linking preserves the original ordering of key-pairs that are *incomparable*, according to our current knowledge, i.e. in the partial order described by the heap.

Similarly, the *rank* of a key in the heap input is mapped to the time of insertion in the BST input.

We remark that the described duality between heaps and trees is quite subtle. As we execute the heap algorithm, the individual rotations that make up the BST execution are revealed in an order that is neither the temporal, nor the spatial order; the order in which the BST execution is revealed depends on the internal structure of the input. An intermediate state of one structure, in our connection, corresponds to an *abstract state* of the other structure, where some decisions pertaining to future operations have been committed to, others are still left undecided.

Smooth heaps. Perhaps the most interesting consequence of our connection is a new, simple heap algorithm that we call the *smooth heap*. We show it to be the heap-counterpart of Greedy, which is conjectured to be instance-optimal for BSTs. Interpreting the proven guarantees for Greedy, we obtain a logarithmic bound on the cost of extract-min in smooth heaps, as well as a number of instance-specific upper bounds, not previously known for any heap implementation (e.g. we show that smooth heaps adapt to locality of reference, and to pattern-avoidance in the input). Assuming the conjectured optimality of Greedy, smooth heaps are also optimal in sorting-mode among all stable heap algorithms. Key to our result is a new, non-deterministic interpretation of Greedy which may be of independent interest.

The intriguing aspect of this connection is that although Greedy is rather complicated to implement as an online BST algorithm, the smooth heap is not; it is comparable in simplicity to pairing heaps. There are *several equivalent descriptions* of the smooth heap. In one view, it appears, roughly speaking, as a pairing heap equipped with a “power-of-two-choices” type of heuristic. In another view, it appears as a structure built of nested treaps. The smooth heap is self-adjusting (no auxiliary information), and uses only the standard pointer structure. Furthermore, it bypasses the known lower bounds for pairing heaps, and hence it may be a plausible proposal for addressing [Question 1](#).

We briefly describe the smooth heap data structure, viewing it in this section as a single multiary (min-)heap, i.e. the key of each node is greater than its parent. (Later a slightly different, forest-based implementation is developed.)

The only operation with a non-trivial implementation is extract-min. Here, after deleting the root, we are left with a list of nodes that need to be consolidated into a single tree, using link operations. We proceed by repeatedly finding a *local maximum* (i.e. an item that is larger than its neighboring siblings). We then link the local maximum with the *larger* of its two neighbors (or with its only neighbor, if it is the leftmost or the rightmost item in the list). This operation, in effect, removes the local maximum, “smoothing” the sequence of items; this gives the name of the data structure. Linking with the larger of the two neighbors can be seen as a locally greedy choice: as we are moving towards the sorted order, intuitively the smaller the rank-difference between linked items, the more progress we make.

A high-level description is given as [Algorithm 1](#). More explicit descriptions are given in [§ 4](#). In [Algorithm 1](#), the described comparison assumes missing nodes (i.e. *null*) to have key value $-\infty$.

Algorithm 1: Smooth heap re-structuring for extract-min.

```

Input: Input: top-level list of roots  $X = (x_1, \dots, x_k)$ 
while  $|X| > 1$ :
  let  $x$  be any local maximum in  $X$ 
  if  $x.\text{prev.key} > x.\text{next.key}$ 
    link( $x.\text{prev}, x$ )
  else
    link( $x, x.\text{next}$ )

```

Locating local maxima and performing the links can be done in a single left-to-right pass (similarly to the first pass of pairing heaps). The remaining top-level items after the first pass are in sorted order, they can therefore be collected in a second, right-to-left accumulation pass (again, similar to the second pass of pairing heaps, except that here, no more comparisons need to be made). The total number of comparisons is easily seen to be at most twice the number of link operations performed. See [Figure 3](#) for an illustration. We remark that the link operations used in the smooth heap are all stable links.

Consequences of the transformation. Several standard heap algorithms can be transferred to the stable heap model, including all natural variants of pairing heaps. Through our duality between heap and tree algorithms, these new heap algorithms also yield new *offline* BST algorithms that have not been previously described.

In the other direction, we can transfer all instance-specific *lower bounds* [[11](#), [65](#)] from BSTs to heaps. For heaps, lower bounds have so far been studied in the *worst-case* setting [[21](#), [36](#), [37](#)], and a theory of instance-specific lower bounds has not yet been proposed (in any heap model). As also observed by Pettie [[54](#)] “*Despite the connection between splay trees and pairing heaps, there is no obvious analogue of dynamic optimality for priority queues.*”

By our results, for *every* stable heap algorithm, there is a corresponding BST algorithm with the same asymptotic cost. This yields a theory analogous to dynamic optimality, for heaps (at least with the restrictions implied by the stable heap model).

Adaptive sorting. Our connection between self-adjusting BSTs and heaps can be interpreted as an equivalence between broad classes of selection-sort and insertion-sort algorithms on inverted input permutations. Sorting has been extensively studied, not just in the worst-case, but also in an instance-specific sense. (Sorting with some “pre-sortedness” in the input is often called “adaptive sorting”). As an entry point into the vast literature on adaptive sorting, we refer to [[2](#), [19](#), [45](#), [46](#), [48–50](#), [52](#)], [[42](#), § 5.3], and references therein.

The proven instance-specific properties of Splay and Greedy-Future subsume many of the structural properties in the adaptive sorting literature. Until now, however, there has been no easy way to implement GreedyFuture as an insertion-sort algorithm. (As mentioned, the online Greedy algorithm is rather complicated.) By using smooth heaps, we obtain an easy-to-implement selection-sort algorithm⁵ that inherits all the instance-specific properties of Greedy (on the inverse of the input permutation).

⁵We remark that *every* stable heap algorithm yields a *stable sorting* algorithm, in the classical sense, assuming that ties are broken according to the “left is smaller” rule.

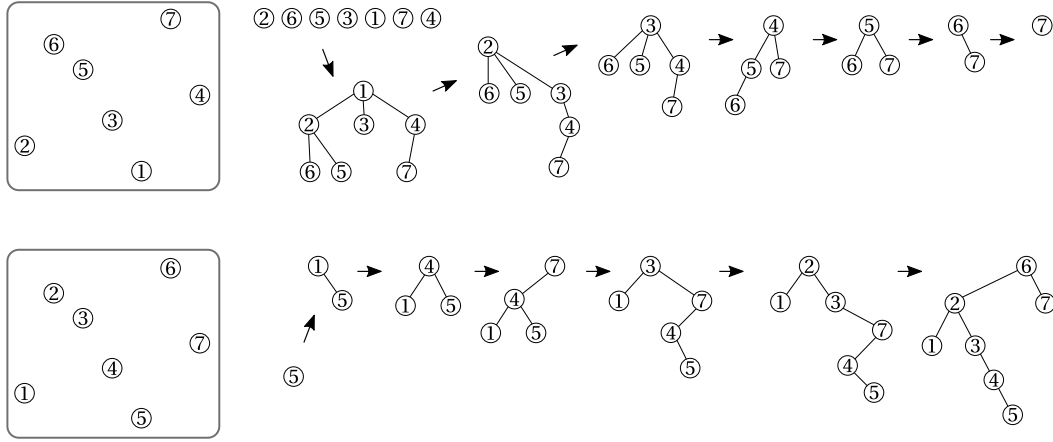


Figure 2: Sorting-mode execution for heaps and BSTs. Input permutation $P = (2, 6, 5, 3, 1, 7, 4)$ and its inverse $P' = (5, 1, 4, 7, 3, 2, 6)$. Above left, P shown with values on y -axis, below left, P' shown with values on x -axis. Above: sequence of extract-mins in a heap, following insertion sequence P , using an unspecified stable heap algorithm. Below, insertion sequence P' in an initially empty BST, using the “move-to-root” restructuring (the inserted items are rotated to the root).

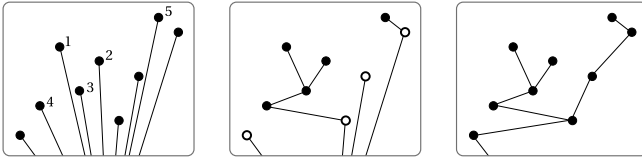


Figure 3: Smooth heap re-structuring. Top-level siblings, left-to-right with keys $(1, 3, 7, 4, 6, 2, 5, 9, 8)$, key values shown on y -axis, subtrees not shown. (i) Initial state. Numbers indicate order of linking. (ii) State after left-to-right smoothing round. Remaining top-level nodes appear hollow. (iii) State after final, right-to-left round. Lowest node is new root after extract-min.

Related work. A “dual” view of binary search trees and heaps brings to mind the Cartesian tree data structure [63], also known as treap [57]. A treap is a binary tree built over pairs of values, respecting the in-order condition with respect to the first, and the heap-order condition with respect to the second entry. Assuming distinct priorities, the treap is easily seen to be unique.

The operation performed by smooth heaps on a list of items can, in fact, be seen as a linear-time “on-the-fly treapification”, where the left-to-right indices of items play the role of key-values.

Such a description of the smooth heap is reminiscent of the Cartesian tree sorting algorithm of Levkopoulos and Petersson [45]. There too, a treap of the input permutation is built. Afterwards, a secondary heap structure (e.g. a binary heap or a Fibonacci-heap) is used to store the children of the root; in this secondary heap, the minimum is repeatedly deleted and its children inserted, resulting in a sorted deletion-order. Cartesian tree sorting was not proposed as a general-purpose data structure, but it could be turned into one, by implementing insert and other operations. (A natural way to insert is perhaps to add a new key into the auxiliary heap.)

Despite the apparent similarity, the behavior and performance of smooth heaps and Cartesian tree sorting are different (see [43] for details).

2 BST MODEL

We adopt a standard model of binary search trees (BSTs) from the literature (see e.g. [11, 12, 47, 65]), which can be seen as a restricted pointer machine model. Each node of the BST is associated with a key, respecting the in-order condition (each item is larger than those in its left subtree and smaller than those in its right subtree). Dictionary operations are performed using a cursor that points, at the beginning of every operation, to the root of the tree. The elementary (unit-cost) operations allowed are moving the cursor to the parent, left-child, or right-child of the current node, and performing a rotation on the edge between the node at the cursor and its parent (Figure 1).

In this paper we consider only search and insert operations; we further assume that all searches are successful. Throughout, we assume that all key-values are distinct.⁶

Search and insert are implemented in the standard way. For search, it is required that the cursor visits, at some point during the operation, the node with the given key. For insert, the cursor must visit both the predecessor and the successor of the new key (or only one of them, in case the key is the new minimum or maximum). A node containing the new key is then attached as the right child of its predecessor, or as the left child of its successor (whichever slot is free). In addition, during both search and insert, a BST algorithm may perform arbitrary re-arrangement of the tree, using rotations at the cursor. The cost of a search or insert operation is the number of nodes that are *visited* by the cursor (“touched”) during the operation. Observe that the touched nodes form a subtree containing the root. This cost model is easily seen to be equivalent up-to constant factors, with most other reasonable cost models [11, 47, 65].

⁶See [6, 11] for discussion of these standard assumptions.

In this paper we only consider “offline” BST algorithms, i.e. we assume that the entire sequence of insert and search operations is known in advance.⁷ We mainly consider two possible modes of operation: a *search-only mode*, in which, starting from some initial tree a sequence of n distinct searches are performed, and an *insert-only mode*, in which, starting from an empty tree, a sequence of n distinct keys are inserted. (We also call the insert-only mode the *sorting-mode*, as it can be seen as a particular implementation of insertion-sort; the n keys can be read out in sorted order by the in-order traversal of the resulting tree.)

As the operations are performed on distinct keys (in both search-only and insert-only modes), we view a sequence of operations of length n as a permutation over $[n]$. The *execution trace* of a BST algorithm $\mathcal{A}$ serving the sequence X , denoted $\mathcal{A}(X)$, is the set of touched keys for all operations. That is, $\langle i, j \rangle \in \mathcal{A}(X)$, if algorithm $\mathcal{A}$ touches node i when executing the j -th operation of X (“at time j ”). The cost of the execution is $|\mathcal{A}(X)|$, i.e. the total number of nodes touched at all times. For every permutation X over $[n]$, we denote the *offline optimum* in search-only mode and insert-only mode as $\text{OPT}_{\text{bst}}(X)$ and $\text{OPT}_{\text{ibst}}(X)$ respectively, where $\text{OPT}_{\text{bst}}(X) = \min_{\mathcal{A}} \{|\mathcal{A}(X)|\}$ when X is treated as a sequence of search operations, and similarly $\text{OPT}_{\text{ibst}}(X) = \min_{\mathcal{A}} \{|\mathcal{A}(X)|\}$ when X is treated as a sequence of insert operations.

Demaine et al. [11] introduced an elegant geometric view for the study of BST algorithms, which we use throughout the paper. We review some of their results, and slightly extend the model, to handle insertion-sequences.

Satisfied point sets. Let $P \subseteq \mathbb{Z} \times \mathbb{Z}$ be a finite set of points. For every point $p \in P$, we write $p = \langle p.x, p.y \rangle$ where $p.x$ is the x -coordinate of p (the first coordinate) and $p.y$ is the y -coordinate of p (the second coordinate). We denote $P_{x \leq i} = \{p \in P \mid p.x \leq i\}$ and $P_{y \leq i} = \{p \in P \mid p.y \leq i\}$, and similarly, $P_{x=i}$, $P_{x < i}$, and so on. We call $P_{x=i}$ and $P_{y=i}$ the i -th column and the i -th row of P respectively. We say that P is a *permutation* (of size n) if $|P_{x=i}| = |P_{y=i}| = 1$ iff $i \in [n]$.

Given two points $p, q \in P$, we denote as $\square_{pq} \subseteq \mathbb{Z} \times \mathbb{Z}$ the rectangle (possibly degenerate) whose opposite corners are p and q . We say that $\square_{pq}$ is a *satisfied rectangle* if $\square_{pq}$ contains another point $r \in P \setminus \{p, q\}$ (where r can be at a corner or a border of $\square_{pq}$), or if p and q are aligned vertically or horizontally.

A point set P is *satisfied* if, for every pair of points $p, q \in P$, the rectangle $\square_{pq}$ is satisfied. The *satisfied superset problem* asks, given a set of points P , to find a satisfied set $Q \supseteq P$.

Let $\mathcal{A}_{\text{sat}}$ be some algorithm for solving the satisfied superset problem. Given a point set P , we denote by $\mathcal{A}_{\text{sat}}(P)$ the output of $\mathcal{A}_{\text{sat}}$ for input P , i.e. a satisfied superset of P . The *cost* of $\mathcal{A}_{\text{sat}}$ on P is defined to be $|\mathcal{A}_{\text{sat}}(P)|$. Let $\text{OPT}_{\text{sat}}(P)$ be the size of the smallest satisfied superset of P .

The following definition is new.

Definition 2.1 (Insertion-compatible superset). *Given a set of points P , a superset $Q \supseteq P$ is insertion-compatible, if for every $i \in [n]$ we have $\min\{p.y \mid p \in P_{x=i}\} = \min\{q.y \mid q \in Q_{x=i}\}$. In words, Q does not add a point below a point of P .*

We also consider the variant of the satisfied superset problem where the goal is to find an *insertion-compatible* satisfied superset of the input. For every point set P , let $\text{OPT}_{\text{isat}}(P)$ be the size of the smallest insertion-compatible satisfied superset of P . Clearly, $\text{OPT}_{\text{isat}}(P) \geq \text{OPT}_{\text{sat}}(P)$.

Let $X = (x_1, \dots, x_n) \in [n]^n$ be a permutation, i.e. $x_i \neq x_j$ for every $i \neq j$. We call $P^X = \{\langle x_i, i \rangle \mid i \in [n]\}$ the *point set of X* .

Given X , we also consider the *inverse permutation* X' . (The i -th element of X is x_i iff the x_i -th element of X' is i .) Treating P^X as an n -by- n matrix, we can define the transpose $(P^X)'$. Observe that $(P^X)' = P^{X'}$. We also consider the *reverse permutation* X^r . (The i -th element of X^r is x_{n-i+1} .) For a point set $P \subseteq [n] \times [n]$, its reverse P^r is such that $P_{y=i}^r = \{\langle x, i \rangle \mid \langle x, n-i+1 \rangle \in P_{y=n-i+1}\}$ for each i . Note that $(P^X)^r = (P^{X^r})$. Throughout this paper, we usually use X to denote a sequence of numbers, and use P or Q to denote a point set.

THEOREM 2.2 (GEOMETRY OF BSTs, SEARCH-ONLY [11]). *Let $X \in [n]^n$ be an arbitrary permutation. A set $Q \subset [n]^2$ is a satisfied superset of P^X iff there is an offline BST algorithm $\mathcal{A}_{\text{bst}}$ in search-only mode with some initial tree such that $\mathcal{A}_{\text{bst}}(X) = Q$.*

Corollary 2.3. *For every permutation $X \in [n]^n$, $\text{OPT}_{\text{bst}}(X) = \text{OPT}_{\text{bst}}(X') = \text{OPT}_{\text{bst}}(X^r)$.*

The following theorem is new, and its proof closely follows the proof of 2.2 from [11].

THEOREM 2.4 (GEOMETRY OF BSTs, INSERTION-COMPATIBLE). *Let $X \in [n]^n$ be an arbitrary permutation. A set $Q \subset [n]^2$ is an insertion-compatible satisfied superset of P^X iff there is an offline BST algorithm $\mathcal{A}_{\text{bst}}$ in insert-only mode such that $\mathcal{A}_{\text{bst}}(X) = Q$.*

Remark 2.5. By Theorems 2.2 and 2.4, given any BST algorithm $\mathcal{A}$ in insert-only mode (a.k.a. sorting-mode), there is a BST algorithm $\mathcal{A}'$ in search-only mode, such that $|\mathcal{A}'(X)| \leq |\mathcal{A}(X)|$ holds for all permutations X . That is, an upper bound for the insert-only mode is also an upper bound for the search-only mode. It is not known whether the reverse statement holds.

GreedyFuture and Greedy. Perhaps the most natural algorithm for the satisfied superset problem is Greedy, a straightforward geometric sweepline algorithm.

Definition 2.6 (Greedy [11]). *Given a set $P \subseteq [n] \times [n]$, Greedy works in n steps as follows. Initially, $Q = P$. At the i -th step, if there is a point $p \in P_{y=i}$ and $q \in Q_{y < i}$ where $\square_{pq}$ is not satisfied, it adds a point $r = \langle q.x, i \rangle$ to Q (so now $\square_{pq}$ is satisfied). The output of Greedy is denoted $\text{Greedy}(P) = Q$.*

Observe that Greedy always produces an insertion-compatible satisfied superset. It is known [11] that Greedy is equivalent, in the sense of Theorem 2.2, to the offline BST algorithm GreedyFuture, i.e. for all permutations X it holds that $\text{GreedyFuture}(X) = \text{Greedy}(P^X)$. Similarly to Splay [58], GreedyFuture is conjectured to be instance-optimal.

Conjecture 2.7 ([11, 47, 51]). *For all permutations $X \in [n]^n$ it holds that $\text{GreedyFuture}(X) = O(\text{OPT}_{\text{bst}}(X))$.*

⁷For online algorithms, operations are revealed one by one.

3 STABLE HEAP MODEL

In this section we introduce a family of heap data structures that include close variants of several of the previously proposed heaps. The key feature of our model is the stable link operation that replaces the standard (unstable) link (Figure 1). We call the new model the stable heap model.

3.1 Description of the Model

Structure. A heap in the stable heap model is organized as a *forest* of multiary min-heaps.⁸ Every node has an associated key (we refer interchangeably to a node and its key), and every non-root key is larger than its parent. For simplicity, we assume keys to be distinct. Internally, a forest of multiary heaps is represented in the standard way by storing the children of every node as a linked list, see e.g. [41, § 2.3.2]. The collection of top-level roots in the forest is also stored in a linked list. We assume that all lists are *doubly-linked*, providing the pointers *next* and *prev* between neighboring siblings, with the appropriate markers at the two ends of the list. We further assume that every node has a pointer to its *leftmost* and to its *rightmost* child, and that we have a global pointer to the *leftmost* and *rightmost* among the top-level roots. The presence of parent pointers is not assumed.

In practice, to make the heap more space-efficient, it is desirable to have only two pointers per node, instead of four, as described. To this effect, algorithms in the stable heap model can also be implemented using only one of *prev* or *next*. Furthermore, one of the pointers *leftmost* and *rightmost* can be simulated in constant time by making the lists circularly linked. The discussion of similar issues of implementation by Fredman et al. [23] for pairing heaps also applies to our model.

Stable link. The defining feature of the stable heap model is the stable link operation (1). We denote by $\text{link}(x)$ the operation of linking x and its right sibling $y = x.\text{next}$ with a stable link. If the key of x is smaller than the key of y , then y becomes the rightmost child of x . Otherwise, x becomes the leftmost child of y . (Contrast this again with the standard, “unstable”, link operation where the larger item always becomes the leftmost child of the smaller.) Clearly, all necessary pointer changes can be performed in constant time. With a careful implementation, the stable link has, in practice, similar cost as the standard link.

In the following we assume (without explicitly describing the low-level details) that the link operations correctly update all pointers to reflect the structure-change.

Operations. The heap operations *makeheap*, *insert*, *decrease-key*, and *meld* can be implemented in a straightforward way, identically to pairing heaps, apart from our use of stable links instead of standard links.

Makeheap creates a new, empty heap with the structure described above. *Insert* creates a singleton root with the new key, and appends it to the list of top-level roots of the heap. *Melding* two heaps concatenates their top-level root-lists. *Decrease-key* detaches

⁸We opt for a forest-based representation in order to simplify the implementation of insert, meld, and decrease-key operations, which can now be performed lazily. It is not hard to change our model such that it is based on a single heap. Fredman [22] describes a general transformation that turns various heap implementations into the forest-of-heaps representation.

the tree rooted at the node whose key is decreased and appends it to the top-level root list.⁹ All four operations require constant number of pointer moves and pointer changes.

We now describe *extract-min*, the most complex operation. To find the minimum, the roots in the top-level list are consolidated into a single tree, through a sequence of stable links. In our model *only neighboring siblings* can be linked. Every link operation removes one root from the top-level list (the one with the larger key of the two linked). Thus, the number of links performed during *extract-min* is exactly one less than the initial size of the top-level list. After a single top-level root remains (the minimum), it is deleted, and its list of children becomes the new top-level list.

Algorithms in the stable heap model differ only in the order in which they link items during the *extract-min* operation. At the start of *extract-min*, a cursor is assumed to point to the leftmost node in the top-level list. Algorithms are allowed to move the cursor to the right or to the left, make comparisons on keys of visited nodes, and perform stable links at the cursor.

We call algorithms in the stable heap model *stable heap algorithms*, and we call the entire structure a *stable heap data structure*.

Cost model. We define the cost of operations to be the *link-only cost*, i.e. the number of stable link operations performed. Thus, the worst-case cost of operations other than *extract-min* is constant, whereas the cost of *extract-min* equals the number of top-level nodes before the operation.

It may seem unrealistic to ignore the cost of comparisons and pointer moves. We justify this choice as follows: (1) In all algorithms that we consider, the number of pointer moves and comparisons will in fact be proportional to the link-only cost, therefore, the use of link-only cost is accurate for these algorithms. (2) We can prove *lower bounds* even for the link-only cost, that is, our lower bounds hold even if pointer moves and comparisons are free. In particular, our lower bounds hold even if the outcomes of all comparisons are known in advance.¹⁰

In the above description, we only link nodes at the top level. This is only for simplicity, and our lower bounds also apply to algorithms that can link siblings at any level of the heap.

Sorting-mode. In this paper we look at stable heap algorithms in *sorting-mode* (see § 1), and analyse the amortized cost of smooth heap operations in this mode only. That is, we assume that a *makeheap* operation is followed by n insert operations with distinct keys, and finally, by n *extract-min* operations.¹¹

Algorithms. We briefly mention four existing algorithms for heap re-structuring that can be adapted to our model, if stable links are used instead of the standard link operation.

⁹The easiest choice is to append the node at one of the ends of the top-level list. An alternative implementation (more in the spirit of stable heaps) would be to “sift-up” the node with decreased key, placing it between its successor and predecessor in the top-level list, according to the insertion-times.

¹⁰It may seem that knowing the sorted order of keys, a stable heap algorithm can arrange the nodes into a path with a linear number of links, contradicting the information-theoretic lower bound for sorting. Observe however, that we are limited to linking neighboring siblings, which removes the contradiction.

¹¹While *sorting-mode* is somewhat restrictive, we remark that the complexity of classical pairing heap variants such as the front-to-back or multipass heuristics is not known, even in *sorting-mode* [13, 23].

In a **simple** (folklore) heap, the roots are collected in a left-to-right accumulation round, repeatedly linking the (current) leftmost item with its right neighbor. It is easy to construct examples where the amortized cost of extract-min operations is linear, i.e. prohibitively large (e.g. 1, 2, 3, ... for the stable variant and 1, 3, 5, ..., 6, 4, 2 for the unstable variant).

The **standard pairing heap** works in two passes: a left-to-right pairing round in which neighboring items are linked in pairs and a subsequent right-to-left accumulation round. The amortized cost of operations is $O(\log n)$ [23]. In the “**front-to-back**” variant of pairing heaps the second round is performed left-to-right, rather than right-to-left. In the “**multipass**” variant of pairing heaps pairing rounds (identical to the first round of the standard and front-to-back variants) are repeatedly executed, until a single root remains. The current upper bounds for the last two approaches are not known to be tight. We refer to [23, 43] for more details.

3.2 Stable Heaps in Sorting-mode

Let $X \in [n]^n$ be a permutation sequence. We consider the *execution trace* of a stable heap algorithm for n extract-min operations, after inserting the keys in $[n]$ in the order given by X into an initially empty heap (i.e. in sorting-mode).

For a stable heap algorithm $\mathcal{A}$, we denote the set of links performed during the sorting-mode execution of $\mathcal{A}$ on X as $\mathcal{A}(X)$, i.e. $(a, b) \in \mathcal{A}(X)$ if, at some point during the execution of $\mathcal{A}$ on X , the nodes with keys a and b are linked. Observe that a pair of nodes can be linked only once during the execution (once a node is in the subtree of the other, it stays there). Thus, the cost of the execution is $|\mathcal{A}(X)|$. We denote by $\text{OPT}_{\text{stable}}(X)$ the minimum cost of a stable heap algorithm when serving X in sorting-mode.

In the following we describe the combinatorial *star-path problem* as an intermediate step towards proving the formal connection between the stable heap model and the BST model (§ 5).

Recall that a point set $P \subseteq [n] \times [n]$ is a permutation if $|P_{x=i}| = |P_{y=i}| = 1$ for each i . In this case, we denote as $p_{y=i}$ the unique point in $P_{y=i}$, and similarly for $p_{x=i}$. Let P_0 denote the set $\{(0, 0)\} \cup P$, i.e. the set P augmented with the origin.

We consider trees with nodeset P_0 . We call such a tree a *monotone tree*, if $\langle 0, 0 \rangle$ is the root, and for every edge (u, v) of the tree, $u.y < v.y$ iff u is closer to the root (in graph-theoretical sense) than v .

Two particular monotone trees are important: $\text{star}(P)$ is the tree in which every point in P is the child of $\langle 0, 0 \rangle$, and $\text{path}(P)$ is the path $\langle \langle 0, 0 \rangle, p_{y=1}, p_{y=2}, \dots, p_{y=n} \rangle$. See Figure 4 for illustration.

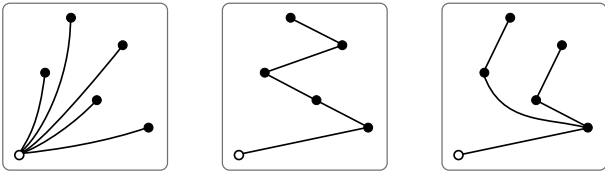


Figure 4: Sequence $X = (3, 5, 2, 4, 1)$ and corresponding point set $P = P^{X'}$ (values shown on y -coordinate). (i) $\text{star}(P)$, (ii) $\text{path}(P)$, (iii) another monotone tree with nodes $P \cup \{(0, 0)\}$.

We define a link operation in monotone trees as follows. Let a and b be two *neighboring* siblings in the tree (by x -coordinate), and

let u be their parent. Suppose $a.y > b.y$. Then $\text{link}(a, b)$ deletes the edge (u, a) and adds the edge (b, a) (i.e. changes the parent of a from u to b). Otherwise, if $a.y < b.y$, we delete (u, b) and add (a, b) . (See Figure 5.)

A link can be performed on any monotone tree that has a node with at least two children, e.g. $\text{star}(P)$. Starting from $\text{star}(P)$, after at most $O(n^2)$ links we reach $\text{path}(P)$, and no more links are possible. (To see this, we can argue that the total length of the y -components of edges decreases with every link.)

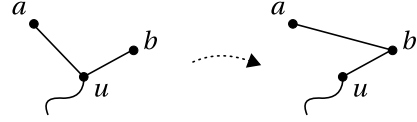


Figure 5: Operation of linking a and b when $a.y > b.y$.

Definition 3.1 (Star-path problem). *Given a permutation point set P , transform $\text{star}(P)$ into $\text{path}(P)$ using a sequence of link operations.*

For a point set P , we denote by $\mathcal{B}(P)$ the set of links performed during the execution of an algorithm $\mathcal{B}$ for the star-path problem with input P , i.e. $(a, b) \in \mathcal{B}(P)$, if at some point, $\mathcal{B}$ links a and b (again, observe that this can happen at most once). We show that stable heap executions for a permutation sequence X and star-path executions for point set $P^{X'}$ (from $\text{star}(P^{X'})$ to $\text{path}(P^{X'})$) are, in a precise sense, equivalent.

THEOREM 3.2. *Let $X = (x_1, \dots, x_n)$ be an arbitrary permutation of $[n]$, and let $\mathcal{P} \subset [n]^2$. Then there is a stable heap algorithm $\mathcal{A}$, such that $\mathcal{A}(X) = \mathcal{P}$, iff there is an algorithm $\mathcal{B}$ for the star-path problem, such that $\mathcal{B}(P^{X'}) = \mathcal{P}$.*

PROOF. Let $\mathcal{A}$ be an arbitrary stable heap algorithm executed in sorting mode for X . After inserting X , the top-level roots are $x_1, \dots, x_n$ (in this order). We assume these roots to be the children of a virtual root with key 0. Thus, at this stage, the heap structure is exactly $\text{star}(P^{X'})$, i.e. the i -th child of the root is $\langle i, x_i \rangle$. Furthermore, we assume that after each extract-min, the old root is “kept around”, considering the children of the old root the new top level roots. Thus, after n extract-min operations, the structure of the heap is exactly $\text{path}(P^{X'})$, i.e. the sorted path of X , with a dummy root in front. These adjustments to $\mathcal{A}$ are only for convenience, and do not affect its behavior.

To show the equivalence between the two executions, we identify a node with key x_i in the heap with the point $\langle i, x_i \rangle$ in the star-path problem, and we describe a bijection between link operations in the two executions. We maintain the following two invariants:

(Invariant 1): The current heap state is the same as the current monotone tree in the star-path execution (with the virtual roots added in both cases). The keys in the heap view correspond to the y -coordinates in the star-path view, and the left-to-right ordering of siblings in the heap view corresponds to the left-to-right ordering by x -coordinates in the star-path view.

(Invariant 2): For every node q in the monotone tree, we have $\text{parent}(q).\text{prev}.x < q.x < \text{parent}(q).\text{next}.x$. Here, prev and next denote the left and right neighboring siblings of a node (by x -coordinate) in the monotone tree. For convenience we assume that

if q has no left neighbor, then $q.\text{prev}.x = -\infty$, and if q has no right neighbor, then $q.\text{next}.x = +\infty$.

Initially, in the case of a star, both invariants clearly hold. We need to show that linking maintains these invariants.

Consider a link between a and b with parent u , where a is to the left of b . Assume $a.y > b.y$, as in Figure 5. After the link, a becomes the child of b .

In the heap, by Invariant (1), the corresponding items x_a and x_b are neighboring siblings ($a < b$), and $x_a > x_b$. Therefore, linking x_a and x_b is a valid operation and x_a becomes the leftmost child of x_b . (Conversely, if x_a and x_b are neighboring siblings in the heap, then linking a and b is a valid star-path link.)

We need to show that a becomes the leftmost child of b , and thus, the ordering of siblings is by x -coordinate. By Invariant (2), before the link operation for all children c of b we have $c.x > \text{parent}(c).\text{prev}.x = a.x$. Thus, Invariant (1) is maintained.

We need to show that Invariant (2) is not violated. This could only happen if, after the link operation, $a.x$ were smaller than $b.\text{prev}.x$. But this is impossible, since the left neighbor of b is the earlier left neighbor of a . The case $a.y < b.y$ is symmetric, and omitted. ■

We stress that in the star-path problem only the set of edges changes during the execution, the locations of points remain the same. In order to maintain this simple geometric model, the use of stable links is essential. (The classical link operation would move entire subtrees from one place to another.) In the remainder of the paper we view stable heap executions mostly in the “geometric view” of the star-path problem.

4 THE SMOOTH HEAP

In this section we describe the smooth heap, our new heap data structure. The smooth heap conforms to the stable heap model described in § 3 and is based on a forest-of-heaps representation. The implementations of makeheap, insert, decrease-key and meld are those from the description of stable heap algorithms in § 3.

The crucial operation is the restructuring of the top-level list of nodes during extract-min. We give three *equivalent* descriptions of this operation; a *non-deterministic* description (Algorithm 1), a *treap-based* description (Algorithm 2), and a *two-pass* description (Algorithm 3).

Algorithm 2: Smooth heap (*treap* view)

Input: Input: top-level list of roots $X = (x_1, \dots, x_k)$
Transform X into a treap with keys $(1, 2, \dots, k)$ and priorities $(x_1, \dots, x_k)$.

Non-deterministic view. In the top-level list X of items, we repeatedly find an arbitrary *local maximum* x . A local maximum is an item x that is larger than both its neighboring siblings, or, in case x is the leftmost or rightmost item, larger than its only neighbor.

Recall that the operation $\text{link}(y)$ performs a stable link between y and its right neighbor $y.\text{next}$.

We link x with the *larger* of its two neighboring siblings (or its only sibling, in case it is the leftmost or rightmost item). If x had two neighbors, we also refer to this step as a *two-choice event*. As x becomes the child of one of its neighbors, it drops out of X , reducing

Algorithm 3: Smooth heap (*two-pass* view)

Input: Input: top-level list of roots $X = (x_1, \dots, x_k)$
LTR-Smoothing-Round:
while there is x in X s.t. $x.\text{key} > x.\text{next}.key$:
 let x be the *leftmost* such node
 if $(x.\text{prev} = \text{null})$ or $(x.\text{prev}.key < x.\text{next}.key)$
 $\text{link}(x)$
 else
 $\text{link}(x.\text{prev})$
RTL-Accumulate-Round:
while $|X| > 1$:
 $x \leftarrow \text{rightmost}(X).\text{prev}$
 $\text{link}(x)$

the size of X by one. As long as X has at least two elements, there is a suitable next choice of x (for instance, the global maximum of X). When X becomes a singleton, we are done; as this item is the minimum, it can be deleted. The non-deterministic view is useful in analysing smooth heaps, because of its resemblance to our non-deterministic view of Greedy (§ 6).

Two-pass view. This description differs from the non-deterministic description only in the choice of the local maximum x : we always choose the *leftmost* such item. Observe that if the only remaining local maximum is the rightmost item, then the items in the list are *sorted*. In this case, the remaining items can be linked in a single right-to-left pass, which we can execute without further comparisons. It is thus convenient to view the execution in two passes that resemble the description of pairing heaps: a left-to-right *smoothing pass* followed by a right-to-left accumulation pass. The two-pass view of smooth heaps is perhaps the most convenient to implement. A more explicit implementation is given in Algorithm 4.

Treap view. We associate each item x_i in X with a pair of values (i, x_i) , and we transform the list into a treap over the pairs (using an arbitrary method for treap-building). Recall that a treap is a binary tree with a pair of values in every node, respecting the in-order (i.e. search tree order) according to the first entry, and the min-heap order according to the second entry of every pair. As mentioned, such a tree is unique. (The item with the unique minimum priority is the root, and the items with smaller, resp. larger key values form its recursively-built left and right subtrees). We use the treap view in order to connect the non-deterministic and two-pass views of the smooth heap.

A remark is in order: as a treap is a binary tree, each node may have a left and a right child. For every node in X , its left child in the treap will become its *leftmost* child in the underlying tree, and its right child in the treap will become its *rightmost* child. In other words, the existing children of x_i end up *between* the two new children possibly gained during the treap-building.

THEOREM 4.1. *The non-deterministic, two-pass, and treap-based descriptions of smooth heaps describe the same transformation.*

We show that the non-deterministic algorithm constructs a treap regardless of the order in which local maxima are chosen, and thus

the non-deterministic and treap-based descriptions are equivalent. As the two-pass description is just a particular way of choosing the local maxima, it follows that it also produces a treap, and since the treap is unique, all three views are equivalent.

To this end, let T be the tree built from X through repeatedly linking local maxima of X with their larger neighbor, until a single item remains (according to Algorithm 1). The following three properties of T together imply that T is a treap (we omit the proof in this version of the paper).

- (1) T is a *heap* according to the key-values x_i ,
- (2) T is a *binary tree*,
- (3) T is a *search tree*, according to the indices i .

Although there are several linear-time algorithms known for constructing a treap from an array, e.g. [25], we are not aware of a previous mention in the literature of the particular method implicit in the description of the smooth heap. Due to its simplicity, we find it interesting in its own right.

Remark: As mentioned, the smooth heap is implemented using stable links. In the non-deterministic and two-pass descriptions it is in fact also possible to use the classical, unstable link. In this case, however, the three descriptions are no longer equivalent. It is an interesting open question how efficient the resulting unstable two-pass algorithm is.

It is instructive to compare smooth heaps and pairing heaps. Besides the implementation of the link operation (stable or unstable) there are several differences between the two algorithms.

In pairing heaps, as in Fredman's model of generalized pairing heaps, comparisons only occur within a link operation, i.e. a comparison is always followed by the corresponding link. By contrast, in smooth heaps, comparisons are decoupled from links, and are also used to decide *which pair of nodes* to link. In fact, comparisons are used *only* for this purpose: it can be observed in the description of Algorithms 3 and 4 that at the time of linking, it is always already known, which of the two linked items is greater.

Recall that in our model, the cost of the algorithm is its “link cost”, i.e. the number of link operations performed. It can be observed in Algorithm 4 that the number of other elementary operations is proportional to the link cost. In particular, every comparison is followed by moving the cursor to the right, or by a link at the cursor. It follows that the number of comparisons is at most twice the number of links. (The number of items to the right of the cursor *plus* the total number of items decreases after every comparison.)

The distinctive feature of smooth heaps is their *power-of-two-choices* linking, whereby an item x is linked with the larger of its two neighbors (both of which are smaller than x). In other words, of the two possible edges we could create, we choose the one with smaller rank-difference. Intuitively, with this choice, we expect to move closer to the totally ordered state (i.e. a path in which the rank-difference along every edge is one). Depending on the subtrees of the nodes, this choice may, of course, not be globally optimal [43]. On the other hand, as shown in § 5, there is reason to believe that the smooth heap is not far from being optimal; a large gap from the optimum would disprove the conjectured optimality of Greedy.

Finally, we remark that smooth heaps (in the two-pass view) can also be implemented with only two pointers per node add the cost of some loss in simplicity. We refer to [43] for details.

Algorithm 4: Smooth heap re-structuring

Input: A list of nodes, with x initially pointing to the leftmost node

```

(LTR) while  $x.next \neq \text{null}$  do
    if  $(x.key < x.next.key)$  then  $x \leftarrow x.next$ 
    else
        ▷ Local maximum found
        while  $(x.prev \neq \text{null})$  do
            if  $(x.prev.key > x.next.key)$  then
                 $x \leftarrow x.prev$ 
                link( $x$ )
            end
        else
             $x \leftarrow x.next$ 
            link( $x.prev$ )
            continue (LTR)
        end
    end
    link( $x$ )
end

(RTL) while  $x.prev \neq \text{null}$  do
     $x \leftarrow x.prev$ 
    link( $x$ )
end

```

5 CONNECTION AND CONSEQUENCES

In this section we present the main connections between stable heap algorithms and BST algorithms and we discuss the consequences of these connections.

We show that the cost of smooth heaps in sorting-mode, once the role of key-space and time are swapped, matches the cost of GreedyFuture, conjectured to be instance-optimal in the BST model.

THEOREM 5.1 (SMOOTH-GREEDY TRANSFORMATION). *For every permutation X , we have $|GreedyFuture(X')| = \Theta(|Smooth(X)|)$.*

As an immediate consequence of Theorem 5.1 and the amortized analysis of GreedyFuture [20], we obtain.

Corollary 5.2 (Amortized analysis of smooth heaps). *For every permutation $X \in [n]^n$, $|Smooth(X)| = O(n \log n)$.*

For the amortized cost of GreedyFuture several stronger, instance-specific bounds are known. We describe two from the literature.

The *weighted dynamic finger* $WDF(X)$ of an input sequence X is a quantity that describes its *locality of reference*. It was introduced in [35], and it subsumes the earlier *dynamic finger* bound [9, 10, 58], as well as other bounds (see e.g. [7]). For the exact definition of $WDF(\cdot)$ we refer to [35].

A permutation $X \in [n]^n$ avoids a permutation $\pi \in [k]^k$ if there is no subsequence of X (not necessarily contiguous) that is *order-isomorphic* to π . (We refer the reader to [29, 39, 40, 53, 56, 61] for more information on this extensively studied property.) As a simple observation, we mention that if X avoids π then X' avoids π' .

The following results are known.

THEOREM 5.3 ([6, 35],). *For every permutation $X \in [n]^n$:*

- $|Greedy(X)| = O(WDF(X))$.
- If X avoids some permutation $\pi \in [k]^k$, then $|Greedy(X)| = n \cdot 2^{\alpha(n)^{O(k)}}$, where $\alpha(\cdot)$ is the slowly growing inverse Ackermann function.

We immediately obtain the following.

Corollary 5.4 (Instance-specific upper bounds for smooth heaps). *For every permutation $X \in [n]^n$:*

- $|Smooth(X)| = O(WDF(X'))$.
- If X avoids some permutation $\pi \in [k]^k$, then $|Smooth(X)| = n \cdot 2^{\alpha(n)^{O(k)}}$.

More generally, we show that there is a general transformation from an arbitrary stable heap algorithm to an offline BST algorithm.

THEOREM 5.5 (GENERAL TRANSFORMATION). *For every stable heap algorithm $\mathcal{A}_{stable}$ there is an offline BST algorithm $\mathcal{A}_{bst}$, such that for every permutation X , the costs of $\mathcal{A}_{stable}$ and $\mathcal{A}_{bst}$ for sorting X are asymptotically the same, i.e. $|\mathcal{A}_{bst}((X')^r)| = \Theta(|\mathcal{A}_{stable}(X)|)$, and in particular, $OPT_{ibst}((X')^r) = O(OPT_{stable}(X))$.*

Observe that the permutation input of $\mathcal{A}_{bst}$ is inverted and reversed. From Remark 2.5, we note again that bounding from above the running time of a BST algorithm in sorting mode is stronger than bounding it in search-only mode.

For search-only mode, we can obtain infinitely many offline BST algorithms from a single stable heap algorithm, all of which have at most a constant factor larger cost. (Here, *infinitely many* is understood as n tends to infinity, i.e. the number of BST algorithms generated depends on the input size n .) In particular, we obtain multiple (non-trivial) offline BST executions that are $O(1)$ -competitive with GreedyFuture.

The proofs of these results are presented in § 6. Below we discuss some of their further consequences.

5.1 Consequences of the Optimality of GreedyFuture

GreedyFuture is widely conjectured to be instance-optimal [11, 47, 51]. In case the conjecture is true, we obtain the following two statements. First, our smooth heap algorithm is an instance-optimal stable heap algorithm (at least) for sorting. Second, the optimal costs of (1) selection-sort with stable heaps (OPT_{stable}), (2) insertion-sort with BSTs (OPT_{ibst}), and (3) searching with BSTs (OPT_{bst}) are the same within some constant factor. Moreover, each of these quantities are invariant under applying inversion and/or reversion to the input.

Without the assumption, we only know that, for every permutation X , $OPT_{ibst}(X) \geq OPT_{bst}(X)$ holds, and $OPT_{stable}(X) \geq \Omega(OPT_{ibst}((X')^r))$ by Theorem 5.5. It is not clear how to prove the

inequalities in the other direction, or how to compare $OPT_{stable}(X)$ and $OPT_{ibst}(X)$.

Corollary 5.6. *Assuming Conjecture 2.7, for every permutation X ,*

- (i) $Smooth(X) = \Theta(OPT_{stable}(X))$,
- (ii) $OPT_{stable}(X) = \Theta(OPT_{ibst}(X)) = \Theta(OPT_{bst}(X))$, and for each model $\in \{stable, ibst, bst\}$,
 $OPT_{model}(X) = \Theta(OPT_{model}(X')) = \Theta(OPT_{model}(X^r))$.

5.2 Dynamic Optimality for Stable Heaps

The theory of instance-specific lower bounds for BSTs is much richer than the corresponding theory for heaps. There are several *concrete* lower bounds known for OPT_{bst} , Wilber's first bound (a.k.a. the “interleave bound”), Wilber's second bound (a.k.a. the “funnel bound”) [65] and the maximum independent rectangle (MIR) bound [11]. (See also [33] for the precise definitions of these quantities.) It is typically easier to reason about these lower bounds than to analyse OPT_{bst} directly. Currently, all BST algorithms shown to be $o(\log n)$ -competitive (i.e. Tango [12], Multi-splay [64] and Chain-splay [26]) have been shown to be competitive with Wilber's first bound. Some connections between these bounds are known. In [11] it is shown that both Wilber's bounds are subsumed by the MIR bound, which is computable by a sweep-line algorithm (intriguingly) similar to Greedy. Harmon [32] also shows that the MIR bound is captured (up to a constant factor) by the optimal cost of a network-design problem called *small Manhattan networks* [27].¹²

For heaps, lower bounds have so far been studied in the *worst-case* setting [21, 36, 37], and a theory of instance-specific lower bounds has not yet been proposed (in any heap model).

Our results connect the two models, and yield an analogous theory for heaps (at least with the restrictions implied by the stable heap model). By Theorem 5.5, for every stable heap algorithm, there is a corresponding BST algorithm with the same asymptotic cost. By this result, all known instance-specific lower bounds are immediately transferred from BSTs to stable heaps.

Corollary 5.7. *Let $W_1(X)$, $W_2(X)$, $MIR(X)$ be Wilber's two bounds [65], and the maximum independent rectangle (MIR) bound [11] for an arbitrary permutation X . Then we have*

$$W_1(X), W_2(X) \leq MIR(X) \leq O(OPT_{stable}(X)).$$

6 PROOFS

We present the main ideas of the proofs of Theorems 5.1 and 5.5 from § 5, deferring some details to [43]. We start by describing a non-deterministic algorithm for the satisfied superset problem whose output is exactly the same as the output of Greedy, described in § 2. We call this algorithm “non-deterministic Greedy” and denote it Greedy_{nondet}. This alternative description of Greedy can be of independent interest; we use it to show a connection between the smooth heap and Greedy. We first define the ADD gadget.

Definition 6.1 (ADD gadget). *Let $P \subseteq \mathbb{Z} \times \mathbb{Z}$ be a point set. We say that $G = (a, b, c, d, e)$ is an ADD gadget in P if*

- $\{a, b, c, d, e\} \subseteq P$

¹²Wilber's first bound is in fact, a family of bounds, as it is defined with respect to a reference tree. Here, by $W_1(X)$ we denote the best bound in the family, i.e. using the reference tree that maximizes the bound for given X .

- The relative positions of a, b, c, d, e are as in Fig. 6 or its horizontal reflection, i.e. $a.y > b.y > c.y = d.y = e.y$ and $e.x < a.x = c.x < b.x = d.x$, or $e.x > a.x = c.x > b.x = d.x$.
- Let $f = \langle b.x, a.y \rangle$. Then $f \notin P$
- Let $\square_G = \square_{ef} \setminus (\{ \langle e.x, i \rangle \mid i \} \cup \{ \langle i, e.y \rangle \mid i \} \cup \{ \langle b.x, i \rangle \mid i \} \cup \{ \langle i, a.y \rangle \mid i \})$. In words, $\square_G$ is $\square_{ef}$ with the borders removed. Then, $\square_G \cap P = \emptyset$. We call $\square_G$ the rectangle inside the gadget G . (Figure 6)

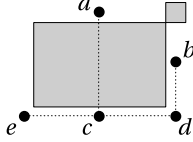


Figure 6: ADD gadget $G = (a, b, c, d, e)$. Larger shaded rectangle is $\square_G$, small square is $\langle b.x, a.y \rangle$. Dotted lines indicate horizontal or vertical alignment. Shaded areas are free of points.

By filling the gadget G we mean the action of adding the point $f = \langle b.x, a.y \rangle$ into a point set P when there is a gadget $G = (a, b, c, d, e)$ in P . We observe that Greedy always fills an ADD gadget.

Lemma 6.2. *If there is an ADD gadget $G = (a, b, c, d, e)$ in P , then $\langle b.x, a.y \rangle \in \text{Greedy}(P)$.*

Moreover:

Lemma 6.3. *Let $Q \subseteq [n] \times [n]$ be an arbitrary point set. There is no ADD gadget in $(Q \cup \text{box})$ if and only if Q is satisfied.*

The above lemma suggest the definition of $\text{Greedy}_{\text{nondet}}$, which we argue to be equivalent with Greedy. Let $\text{box} = \{ \langle i, 0 \rangle \mid i \in [n] \} \cup \{ \langle 0, i \rangle \mid 0 \leq i \leq n \} \cup \{ \langle n+1, i \rangle \mid 0 \leq i \leq n \}$.

Definition 6.4 (Non-deterministic Greedy). *Given a set $P \subseteq [n] \times [n]$, Greedy_{nondet} works as follows. Initially, set $Q = (P \cup \text{box})$. Repeatedly find an arbitrary ADD gadget $G = (a, b, c, d, e)$ in Q . Then, fill G (i.e. add point $f = \langle b.x, a.y \rangle$ to Q). Let Q be the final point set where there is no ADD gadget in Q . Let $\text{Greedy}_{\text{nondet}}(P) = (Q \setminus \text{box})$.*

THEOREM 6.5. *For every point set $P \subset [n] \times [n]$, we have $\text{Greedy}_{\text{nondet}}(P) = \text{Greedy}(P)$.*

6.1 Reduction to Geometric Setting

We state some results about the connection between the two geometric problems (the star-path problem and the satisfied superset problem). Then, we prove that they imply the main results from § 5.

To prove Theorem 5.1, let $\text{Smooth}_{\text{path}}$ be an algorithm for the star-path problem obtained from the smooth heap algorithm using 3.2. We have the following key lemma:

Lemma 6.6. $|\text{Greedy}(P)| = \Theta(|\text{Smooth}_{\text{path}}(P)|)$ for every permutation point set P .

To prove Theorem 5.5, we show the analogous statement:

Lemma 6.7. *For every algorithm $\mathcal{A}_{\text{path}}$ for the star-path problem, there is an algorithm $\mathcal{A}_{\text{sat}}$ for the satisfied superset problem, such that $|\mathcal{A}_{\text{sat}}(P)| = \Theta(|\mathcal{A}_{\text{path}}(P)|)$ for every permutation P . Moreover, $(\mathcal{A}_{\text{sat}}(P))^r$, i.e. the reverse of the output of $\mathcal{A}_{\text{sat}}$, is an insertion-compatible superset of P^r .*

Assuming the two lemmas above, we can prove 5.1 and 5.5.

PROOF. Let $f \approx g$ denote $f = \Theta(g)$. Fix a permutation X on $[n]$. (**Lemma 6.6 implies Theorem 5.1:**)

$$\begin{aligned} |\text{GreedyFuture}(X')| &= |\text{Greedy}(P^{X'})| \\ &\approx |\text{Smooth}_{\text{path}}(P^{X'})| && \text{by 6.6} \\ &= |\text{Smooth}(X)|. && \text{by 3.2} \end{aligned}$$

(**Lemma 6.7 implies Theorem 5.5:**) Given a stable heap algorithm $\mathcal{A}_{\text{stable}}$ from Theorem 5.5, there is $\mathcal{A}_{\text{path}}$ where $\mathcal{A}_{\text{stable}}(X) = \mathcal{A}_{\text{path}}(P^{X'})$ by Theorem 3.2. Plugging $\mathcal{A}_{\text{path}}$ into Lemma 6.7, there is an algorithm $\mathcal{A}_{\text{sat}}$ for the satisfied superset problem, where $|\mathcal{A}_{\text{sat}}(P^{X'})| = \Theta(|\mathcal{A}_{\text{path}}(P^{X'})|)$, with $(\mathcal{A}_{\text{sat}}(P^{X'}))^r$ being an insertion-compatible satisfied superset of $(P^{X'})^r$. By treating $(P^{X'})^r$ as the input, there is an algorithm $\mathcal{A}'_{\text{sat}}$ such that $\mathcal{A}'_{\text{sat}}((P^{X'})^r) = \mathcal{A}_{\text{sat}}(P^{X'})$ and $\mathcal{A}'_{\text{sat}}((P^{X'})^r)$ is insertion-compatible for its input $(P^{X'})^r$. As $(P^{X'})^r = P^{(X')^r}$, there is an offline BST algorithm $\mathcal{A}_{\text{bst}}$ in insert-only mode (sorting mode) such that $\mathcal{A}_{\text{bst}}((X')^r) = \mathcal{A}'_{\text{sat}}(P^{(X')^r})$, by Theorem 2.4. To summarize, we have:

$$\begin{aligned} |\mathcal{A}_{\text{bst}}((X')^r)| &= |\mathcal{A}'_{\text{sat}}(P^{(X')^r})| && \text{by 2.4} \\ &= |\mathcal{A}_{\text{sat}}(P^{X'})| \\ &\approx |\mathcal{A}_{\text{path}}(P^{X'})| && \text{by 6.7} \\ &= |\mathcal{A}_{\text{stable}}(X)|. && \text{by 3.2} \end{aligned}$$

It remains to prove Lemma 6.6 and Lemma 6.7. We use a similar approach, but with different key ingredients in the proof.

6.2 The Common Framework

From now on, we fix an algorithm $\mathcal{A}_{\text{path}}$ for the star-path problem and a permutation P . We describe two different ways of producing a satisfied superset Q of P such that $|Q| = \Theta(|\mathcal{A}_{\text{path}}(P)|)$. The first method works for arbitrary $\mathcal{A}_{\text{path}}$. We show that Q is insertion-compatible for P^r which yields Lemma 6.7. The second method works only for $\mathcal{A}_{\text{path}} = \text{Smooth}_{\text{path}}$. In this case we show that $Q = \text{Greedy}(P)$ which yields Lemma 6.6.

In this subsection, we describe a common framework for constructing a satisfied set Q .

Let $P_0 = P \cup \{ \langle 0, 0 \rangle \}$. Recall the definitions of $\text{star}(P)$ and $\text{path}(P)$ near 3.1 (they are two particular monotone trees whose nodes are P_0). Initially, we set $Q = Q^{\text{init}}$ where the definition of Q^{init} differs in the two cases. For each link that $\mathcal{A}_{\text{path}}$ performs to transform $\text{star}(P)$ to $\text{path}(P)$, we add a constant number of points to Q while maintaining certain invariants. Once the execution of $\mathcal{A}_{\text{path}}$ is finished (producing $\text{path}(P)$), the invariant will imply that Q is satisfied. To state the invariants precisely, we need some definitions.

Let T be the current tree of $\mathcal{A}_{\text{path}}$. For convenience, we write the set of nodes of T as $P_0 \stackrel{\text{def}}{=} \{u_0, \dots, u_n\}$ where $u_0 = \langle 0, 0 \rangle$ and u_i is

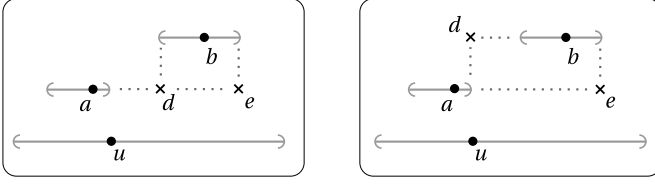


Figure 7: Two alternatives for a step in the general transformation. After linking b to a (with common parent u), points d and e (marked as crosses) are added. Dotted lines show horizontal or vertical alignment. Horizontal intervals are defined such as to minimally cover the points in every row.

the unique point in $P_{y=i}$. Let $Q \supseteq Q^{init}$ be the current point set. For every i , we write $Q_{u_i} = Q_{y=i}$. For every node u we define $I(u) \subset [n]$ as the interval of u and we write $I(u) \stackrel{\text{def}}{=} (I(u).min, I(u).max)$. We call I the interval function. The precise definition of I , which is a key element of the proofs, will be given later.

We have the following invariants.

- **Invariant 1 (Intervals respect tree-structure):** The set of intervals $I(u)$ for all nodes $u \in T$ forms a laminar family whose structure is exactly T . That is, let u be an arbitrary node in T with children $v_1, \dots, v_k$. Then $I(v_j) \subseteq I(u)$ for all j , and $I(v_j) \cap I(v_{j'}) = \emptyset$ for all $j \neq j'$.
- **Invariant 2 (Tree-structure respects satisfiability):** If u is an ancestor of v in T , then for every $p \in Q_u$ and $q \in Q_v$ the rectangle $\square_{pq}$ is satisfied.
- **Invariant 3 (No points between a node and its parent's interval):** For every node v with parent u , we have $\{\langle v.x, i \rangle \mid u.y < i < v.y\} \cap Q = \emptyset$.
- **Invariant 4 (Adding interval endpoints):** For every node v with parent u ,

$$\begin{aligned} \langle I(v).min, v.y \rangle, \langle I(v).max, v.y \rangle &\in Q \text{ (endpoints of } I(v)), \\ \langle I(v).min, u.y \rangle, \langle I(v).max, u.y \rangle &\in \\ &Q \text{ (endpoints of } I(v) \text{ "projected" to row of parent } u). \end{aligned}$$

We make a crucial observation.

Proposition 6.8. *At the end, when $T = \text{path}(P)$, if the invariants hold, then Q is a satisfied set.*

PROOF. Let p and q be two arbitrary points in Q . W.l.o.g. $p \in Q_{u_i}$ and $q \in Q_{u_j}$ where $i \leq j$. As $T = \text{path}(P)$, u_i is an ancestor of u_j . By Invariant (2), $\square_{pq}$ is satisfied. ■

6.3 The General Transformation

Let $\text{base} = \{(i, 0) \mid i \in [n]\}$. We define $Q^{init} \stackrel{\text{def}}{=} P \cup \text{base}$, and define the interval function as follows. For each node u_i ,

$$I(u_i) \stackrel{\text{def}}{=} (\min_{q \in Q_{u_i}} q.x, \max_{q \in Q_{u_i}} q.x).$$

In words, $I(u_i)$ is the open interval between the leftmost and rightmost “touched” positions in the i -th row. With this definition of Q^{init} and I , we observe that the invariants hold initially. Next, we specify how to add points to Q for each link performed by $\mathcal{A}_{\text{path}}$ and show that the invariants are maintained.

Lemma 6.9. *Let T be the current tree and Q be the current point set. Suppose that the invariants hold for T and Q . Let T' be obtained from T by a stable link operation. We can add **one** or **two** new points to Q and obtain Q' such that the invariants hold for T' and Q' .*

We conclude with the proof of Lemma 6.7.

PROOF OF LEMMA 6.7. Given $\mathcal{A}_{\text{path}}$ and P , we let $\mathcal{A}_{\text{sat}}$ be simply the algorithm that returns Q as described above. By Lemma 6.9, for each link of $\mathcal{A}_{\text{path}}$, one or two new points are added to Q . So $|Q| = \Theta(|\mathcal{A}_{\text{path}}(P)|)$. By Lemma 6.8 and the fact that $Q \supseteq (P \cup \text{base})$, Q is a satisfied superset of P . Lastly, it is easy to see from Figure 7(left) that all points added to Q are “below” P , i.e. for every $q \in Q_{x=i}$ we have $q.y \leq (p_{x=i}).y$, where $p_{x=i}$ is the unique in the i -th column of P . (To see this, consider for contradiction the first time a point would be added above a point of P .) In other words, Q is an insertion-compatible superset of P^r . ■

The proof of Lemma 6.6 is deferred to the full version of the paper [43].

ACKNOWLEDGEMENT

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 715672 and through the ERC consolidator grant No 617951, and from the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11), and from the Swedish Research Council (Reg. No. 2015-04659).

REFERENCES

- [1] G. M. Adelson-Velskii and E. M. Landis. 1962. An algorithm for organization of information. *Dokl. Akad. Nauk SSSR* 146 (1962), 263–266.
- [2] Jérémy Barbay and Gonzalo Navarro. 2013. On compressing permutations and adaptive sorting. *Theoretical Computer Science* 513 (2013), 109 – 123. <https://doi.org/10.1016/j.tcs.2013.10.019>
- [3] Rudolf Bayer. 1972. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica* 1, 4 (01 Dec 1972), 290–306. <https://doi.org/10.1007/BF00289509>
- [4] Gerth Stølting Brodal. 2013. A Survey on Priority Queues. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. 150–163. https://doi.org/10.1007/978-3-642-40273-9_11
- [5] Gerth Stølting Brodal, George Lagogiannis, and Robert Endre Tarjan. 2012. Strict fibonacci heaps. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*. 1177–1184. <https://doi.org/10.1145/2213977.2214082>
- [6] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. 2015. Pattern-Avoiding Access in Binary Search Trees. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. 410–423.
- [7] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. 2016. The landscape of bounds for binary search trees. *CoRR* abs/1603.04892 (2016). <http://arxiv.org/abs/1603.04892>
- [8] Timothy M. Chan. 2013. *Quake Heaps: A Simple Alternative to Fibonacci Heaps*. Springer Berlin Heidelberg, Berlin, Heidelberg, 27–32. https://doi.org/10.1007/978-3-642-40273-9_3
- [9] R. Cole. 2000. On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof. *SIAM J. Comput.* 30, 1 (2000), 44–85. <https://doi.org/10.1137/S009753979732699X> arXiv: <http://dx.doi.org/10.1137/S009753979732699X>
- [10] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. 2000. On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting Log n-Block Sequences. *SIAM J. Comput.* 30, 1 (April 2000), 1–43. <https://doi.org/10.1137/S0097539797326988>
- [11] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, and Mihai Patrascu. 2009. The geometry of binary search trees. In *SODA 2009*. 496–505. <http://dl.acm.org/citation.cfm?id=1496770.1496825>
- [12] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. 2007. Dynamic Optimality - Almost. *SIAM J. Comput.* 37, 1 (2007), 240–251. <https://doi.org/10.1137/S0097539705447347>

- [13] Dani Dorfmán, Haim Kaplan, László Kozma, and Uri Zwick. 2017. Pairing heaps: the forward variant. *CoRR* abs/1709.01152 (2017). <http://arxiv.org/abs/1709.01152>
- [14] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. 1988. Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation. *Commun. ACM* 31, 11 (1988), 1343–1354. <https://doi.org/10.1145/50087.50096>
- [15] Amr Elmasry. 2004. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science* 314, 3 (2004), 459 – 466. <https://doi.org/10.1016/j.tcs.2004.01.019>
- [16] Amr Elmasry. 2006. A Priority Queue with the Working-set Property. *Int. J. Found. Comput. Sci.* 17, 6 (2006), 1455–1466. <https://doi.org/10.1142/S0129054106004510>
- [17] Amr Elmasry. 2010. The Violation Heap: a Relaxed Fibonacci-like Heap. *Discrete Math., Alg. and Appl.* 2, 4 (2010), 493–504. <https://doi.org/10.1142/S1793830910000838>
- [18] Amr Elmasry, Arash Farzan, and John Iacono. 2012. A priority queue with the time-finger property. *J. Discrete Algorithms* 16 (2012), 206–212. <https://doi.org/10.1016/j.jda.2012.04.014>
- [19] Vladimir Estivill-Castro and Derick Wood. 1992. A Survey of Adaptive Sorting Algorithms. *ACM Comput. Surv.* 24, 4 (Dec. 1992), 441–476. <https://doi.org/10.1145/146370.146381>
- [20] Kyle Fox. 2011. Upper Bounds for Maximally Greedy Binary Search Trees. In *WADS 2011*. 411–422. https://doi.org/10.1007/978-3-642-22300-6_35
- [21] Michael L. Fredman. 1999. On the Efficiency of Pairing Heaps and Related Data Structures. *J. ACM* 46, 4 (1999), 473–501. <https://doi.org/10.1145/320211.320214>
- [22] Michael L. Fredman. 1999. A Priority Queue Transform. In *Algorithm Engineering, 3rd International Workshop, WAE '99, London, UK, July 19–21, 1999, Proceedings*. 244–258. https://doi.org/10.1007/3-540-48318-7_20
- [23] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica* 1, 1 (1986), 111–129. <https://doi.org/10.1007/BF01840439>
- [24] Michael L. Fredman and Robert Endre Tarjan. 1984. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24–26 October 1984*. 338–346. <https://doi.org/10.1109/SFCS.1984.715934>
- [25] Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. 1984. Scaling and Related Techniques for Geometry Problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA*. 135–143. <https://doi.org/10.1145/800057.808675>
- [26] George F. Georgakopoulos. 2008. Chain-splay trees, or, how to achieve and prove $\log \log N$ -competitiveness by splaying. *Inf. Process. Lett.* 106, 1 (2008), 37–43.
- [27] Joachim Gudmundsson, Oliver Klein, Christian Knauer, and Michiel Smid. 2007. Small Manhattan networks and algorithms for the earth mover's distance. In *In Proc. 23rd European Workshop Comput. Geom. (EWCG'07)*. 174–177.
- [28] Leonidas J. Guibas and Robert Sedgewick. 1978. A Dichromatic Framework for Balanced Trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16–18 October 1978*. 8–21. <https://doi.org/10.1109/SFCS.1978.3>
- [29] Sylvain Guillemot and Daniel Marx. 2014. Finding small patterns in permutations in linear time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5–7, 2014*. 82–101. <https://doi.org/10.1137/1.9781611973402.7>
- [30] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan. 2011. Rank-Pairing Heaps. *SIAM J. Comput.* 40, 6 (2011), 1463–1485. <https://doi.org/10.1137/100785351>
- [31] Thomas Dueholm Hansen, Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. 2015. Hollow Heaps. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6–10, 2015, Proceedings, Part I*. 689–700. https://doi.org/10.1007/978-3-662-47672-7_56
- [32] Dion Harmon. 2006. *New Bounds on Optimal Binary Search Trees*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [33] John Iacono. 2013. In Pursuit of the Dynamic Optimality Conjecture. In *Space-Efficient Data Structures, Streams, and Algorithms*. Lecture Notes in Computer Science, Vol. 8066. Springer Berlin Heidelberg, 236–250. https://doi.org/10.1007/978-3-642-40273-9_16
- [34] John Iacono and Stefan Langerman. 2005. Queaps. *Algorithmica* 42, 1 (2005), 49–56. <https://doi.org/10.1007/s00453-004-1139-5>
- [35] John Iacono and Stefan Langerman. 2016. *Weighted dynamic finger in binary search trees*. Chapter 49, 672–691. <https://doi.org/10.1137/1.9781611974331.ch49> <http://epubs.siam.org/doi/pdf/10.1137/1.9781611974331.ch49>
- [36] John Iacono and Özgür Özkan. 2014. A Tight Lower Bound for Decrease-Key in the Pure Heap Model. *CoRR* abs/1407.6665 (2014). <http://arxiv.org/abs/1407.6665>
- [37] John Iacono and Özgür Özkan. 2014. Why Some Heaps Support Constant-Amortized-Time Decrease-Key Operations, and Others Do Not. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8–11, 2014, Proceedings, Part I*. 637–649. https://doi.org/10.1007/978-3-662-43948-7_53
- [38] Haim Kaplan and Robert Endre Tarjan. 2008. Thin heaps, thick heaps. *ACM Trans. Algorithms* 4, 1 (2008), 3:1–3:14. <https://doi.org/10.1145/1328911.1328914>
- [39] S. Kitaev. 2011. *Patterns in Permutations and Words*. Springer. <http://books.google.de/books?id=jgQHTgR5N60C>
- [40] Donald E. Knuth. 1968. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley.
- [41] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
- [42] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*.
- [43] László Kozma and Thatchaphol Saranurak. 2018. Smooth heaps and a dual view of self-adjusting data structures. *CoRR* abs/1802.05471 (2018). [arXiv:1802.05471](http://arxiv.org/abs/1802.05471)
- [44] Daniel H. Larkin, Siddhartha Sen, and Robert Endre Tarjan. 2014. A Back-to-Basics Empirical Study of Priority Queues. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*. 61–72. <https://doi.org/10.1137/1.9781611973198.7>
- [45] Christos Levcopoulos and Ola Petersson. 1989. *Heapsort—Adapted for presorted files*. Springer Berlin Heidelberg, Berlin, Heidelberg, 499–509. https://doi.org/10.1007/3-540-51542-9_41
- [46] Christos Levcopoulos and Ola Petersson. 1994. Sorting Shuffled Monotone Sequences. *Inf. Comput.* 112, 1 (1994), 37–50. <https://doi.org/10.1006/inco.1994.1050>
- [47] Joan M. Lucas. 1988. Canonical forms for competitive binary search tree algorithms. *Tech. Rep. DCS-TR-250, Rutgers University* (1988).
- [48] Heikki Mannila. 1984. Measures of Presortedness and Optimal Sorting Algorithms (Extended Abstract). In *ICALP 1984*. 324–336. https://doi.org/10.1007/3-540-13345-3_29
- [49] Kurt Mehlhorn. 1979. Sorting Presorted Files. In *Theoretical Computer Science, 4th GI-Conference, Aachen, Germany, March 26–28, 1979, Proceedings*. 199–212. https://doi.org/10.1007/3-540-09118-1_22
- [50] Alistair Moffat and Ola Petersson. 1992. An Overview of Adaptive Sorting. *Australian Computer Journal* 24, 2 (1992), 70–77.
- [51] J.Ian Munro. 2000. On the Competitiveness of Linear Search. In *Algorithms - ESA 2000, Mike S. Paterson (Ed.), Lecture Notes in Computer Science, Vol. 1879*. Springer Berlin Heidelberg, 338–345. https://doi.org/10.1007/3-540-45253-2_31
- [52] J. Ian Munro and Philip M. Spira. 1976. Sorting and Searching in Multisets. *SIAM J. Comput.* 5, 1 (1976), 1–8. <https://doi.org/10.1137/0205001>
- [53] Ilan Newman, Yuri Rabinovich, Deepak Rajendraprasad, and Christian Sohler. 2017. Testing for Forbidden Order Patterns in an Array. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16–19*. 1582–1597. <https://doi.org/10.1137/1.9781611974782.104>
- [54] Seth Pettie. 2005. Towards a Final Analysis of Pairing Heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23–25 October 2005, Pittsburgh, PA, USA, Proceedings*. 174–183. <https://doi.org/10.1109/SFCS.2005.75>
- [55] Seth Pettie. 2008. Splay Trees, Davenport-Schinzel Sequences, and the Deque Conjecture. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1115–1124*. <http://dl.acm.org/citation.cfm?id=1347082.1347204>
- [56] Vaughan R. Pratt. 1973. Computing Permutations with Double-ended Queues, Parallel Stacks and Parallel Queues. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC '73)*. ACM, New York, NY, USA, 268–277. <https://doi.org/10.1145/800125.800408>
- [57] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized Search Trees. *Algorithmica* 16, 4/5 (1996), 464–497. <https://doi.org/10.1007/BF01940876>
- [58] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686. <https://doi.org/10.1145/3828.3835>
- [59] John T. Stasko and Jeffrey Scott Vitter. 1987. Pairing Heaps: Experiments and Analysis. *Commun. ACM* 30, 3 (March 1987), 234–249. <https://doi.org/10.1145/214748.214759>
- [60] Rajamani Sundar. 1992. On the deque conjecture for the splay algorithm. *Combinatorica* 12, 1 (1992), 95–124. <https://doi.org/10.1007/BF01191208>
- [61] Robert Endre Tarjan. 1972. Sorting Using Networks of Queues and Stacks. *J. ACM* 19, 2 (April 1972), 341–346. <https://doi.org/10.1145/321694.321704>
- [62] Robert Endre Tarjan. 1985. Sequential access in splay trees takes linear time. *Combinatorica* 5, 4 (1985), 367–378. <https://doi.org/10.1007/BF02579253>
- [63] Jean Vuillemin. 1980. A Unifying Look at Data Structures. *Commun. ACM* 23, 4 (April 1980), 229–239. <https://doi.org/10.1145/358841.358852>
- [64] Chengwen C. Wang, Jonathan C. Derryberry, and Daniel D. Sleator. 2006. O(Log Log N)-competitive Dynamic Binary Search Trees. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06), Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 374–383*. <http://dl.acm.org/citation.cfm?id=1109557.1109600>
- [65] R. Wilber. 1989. Lower Bounds for Accessing Binary Search Trees with Rotations. *SIAM J. Comput.* 18, 1 (1989), 56–67. <https://doi.org/10.1137/0218004> [arXiv:http://dx.doi.org/10.1137/0218004](http://dx.doi.org/10.1137/0218004)